



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-235388

National Ignition Facility Shot Data Analysis Module Guidelines

S. Azevedo, S. Glenn, A. Lopez, A. Warrick, R.
Beeler

October 9, 2007

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

National Ignition Facility Shot Data Analysis – Module Guidelines

Revision 3.5



Steve Azevedo, Steven Glenn, Aseneth Lopez, Abbie Warrick, Ric Beeler

September, 2007

LAWRENCE LIVERMORE NATIONAL LABORATORY
Lawrence Livermore National Security – Livermore, California – 94551

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This report has been reproduced
directly from the best available copy.
Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62, Oak Ridge, TN 37831
Prices available from (615) 576-8401, FTS 626-8401
Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.,
Springfield, VA 22161

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

National Ignition Facility

Shot Data Analysis – Module Guidelines

Revision 3.5

Prepared by:

Name, Title

Date

Reviewed by:

Name, Title

Date

Name, Title

Date

Name, Title

Date

Approved by:

Name, Title

Date

Table of Contents

1.0 INTRODUCTION	1
1.1 Module Development.....	1
1.2 Manual vs Automatic Operations	2
1.3 Guiding Principles	3
2.0 SETTING UP AN IDL DEVELOPER ENVIRONMENT	3
3.0 IDL ANALYSIS MODULE GUIDELINES.....	5
3.1 Use Functions, not Procedures	5
3.2 Pass Data through the Argument List with Keywords	6
3.3 Provide Test Code and Data	6
3.4 Reduce Duplication	6
3.5 Provide Early Error Notification	6
3.6 Document the Module	7
3.7 Return Metrics.....	7
3.8 Follow Naming Conventions.....	7
3.9 Use the Standard Layout	8
3.10 Use IDL Structures as Needed.....	9
3.11 Avoid Global Variables or Common Blocks	9
3.12 Do Not Use Display Commands	9
3.13 Log Messages	9
3.14 Use Supported Procedures.....	10
4.0 MODULE TESTING	10
4.1 Unit Test	10
4.2 Integration Test (Verification)	10
4.3 Performance Validation	11
5.0 REVISION CONTROL	11

6.0 SUMMARY	12
REFERENCES	12
APPENDIX A	13

List of Figures

Figure 1. Shot Data Analysis Engine, Driver and Modules..	2
Figure 2. Template IDL Module.....	5

List of Acronyms

CDR	Conceptual Design Review
CM	Configuration Management
CDMS	Calibration Data Management System
CMS	Content Management System
CMT	Campaign Management Tool
CVS	Concurrent Versioning System
FDR	Final Design Review
HDF	Hierarchical Data Format
ICCS	Integrated Computer Control System for NIF
IDL	Interactive Data Language
IPT	Integrated Product Team
LP	Laser Performance
NIF	National Ignition Facility
OI	Optics Inspection
PDR	Preliminary Design Review
RI	Responsible Individual
RS	Responsible Scientist
SDA	Shot Data Analysis
SDI	Shot Data Integration
SDV	Shot Data Visualization

For a more complete list of NIF acronyms, see http://www-r.llnl.gov/nif/admin/documents/NIF_acronyms.pdf

1.0 Introduction

This document provides the guidelines for software development of modules to be included in Shot Data Analysis (SDA) for the National Ignition Facility (NIF). An Analysis Module is a software entity that groups a set of (typically cohesive) functions, procedures and data structures for performing an analysis task relevant to NIF shot operations. Each module must have its own unique identification (module name), clear interface specifications (data inputs and outputs), and internal documentation.

It is vitally important to the NIF Program that all shot-related data be processed and analyzed in a consistent way that is reviewed by scientific and engineering experts. SDA is part of a NIF Integrated Product Team (IPT) whose goal is to provide timely and accurate reporting of shot results to NIF campaign experimentalists. Other elements of the IPT include the Campaign Management Tool (CMT) for configuring experiments, a data archive and provisioning system called CMS, a calibration and configuration database (CDMS), and a shot data visualization tool (SDV).

We restrict our scope at this time to guidelines for modules written in Interactive Data Language, or IDL¹. This document has sections describing example IDL modules and where to find them, how to set up a development environment, IDL programming guidelines, shared IDL procedures for general use, and revision control.

1.1 Module Development

We will refer to a “Developer” as anyone who writes and implements Analysis Modules. A Developer could be a member of the SDA software development team (referred to as “SDA Developers”), or they could be Diagnostic Responsible Scientists (RS), Campaign Experimentalists, Responsible Individuals (RI), Diagnostics Engineers, and others. Developers will often perform module development and testing in a local compute environment (we will call the “Desktop” environment) using local data for testing. Then, once it is ready for integration and release, the module is transferred to the SDA development team members who work with the Developer on reviews, validation of results, release schedules and configuration management.

This document is directed mostly toward the module development step, and the standards that will make the integration and release steps go smoothly, which will reduce the time and cost of implementation and long-term maintenance. For completeness, the SDA Developers follow a rigorous software deployment methodology that involves:

- Requirements gathering – Reviewed by RS/RI/Users
- Conceptual Design – Module flow (Visio diagram); Reviewed by RS/RI/Users
- Preliminary Design – Data inputs/outputs (Spreadsheet); Reviewed by RS, Developers
- Final Design – List of functions to develop; Reviewed by RS, Developers
- Development – Implement algorithms and performance tests; Reviewed by RS
- Integration testing – Performed by Developers/Testers
- Final Documentation – Reviewed by all
- Regression test – Deliver data and test procedures for automatic testing
- Release to Production
- Revision Control and Management

¹ IDL is a registered trademark of the Research Systems, Inc. (RSI); see [1] for details.

1.2 Manual vs Automatic Operations

Analysis modules must be capable of running both manually (under direct user control) and automatically (without user intervention during the NIF shot cycle). Modules that process production data automatically after a shot are maintained by the SDA module development team.

For automatic SDA on NIF, the modules will be called by the Shot Data Analysis Engine [3] and Drivers [4]. A block diagram of the automated analysis process is shown in Figure 1. The SDA Engine provides the framework for triggering, sequencing and queuing (scheduling) the analysis module operations, while the Analysis Driver controls the flow of data and results using the NIF Content Management System (CMS). The analysis data flow may involve a group of modules to perform an overall analysis task, such as “analyze Static X-ray Imager data”.

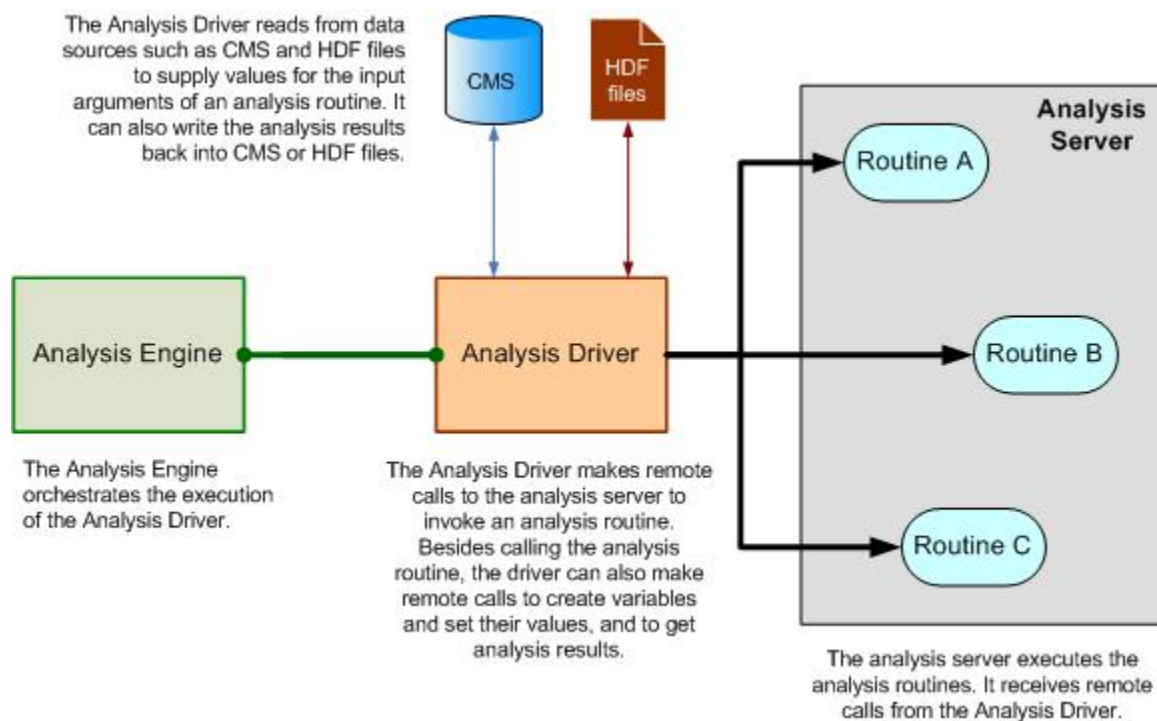


Figure 1. Shot Data Analysis Engine, Driver and Modules. The Engine invokes Analysis Drivers based on an analysis flow model. The Driver accesses the data to/from the Content Management System and initiates the (possibly multi-step) module routines of the modules on an Analysis Compute Server.

The IDL module functions that are directly called by the Drivers (Routines A, B, and C in the figure) will be referred to as “top-level functions”, which may have their own lower level functions or procedures. These top-level analysis functions must be able to accept data from, and return data to, the Analysis Driver and Engine through the argument list. The input data will include configuration data, raw data, calibration data and/or processed data. Output data will include processed data, as well as error, quality, and other information. All of the above data, and the Analysis Modules that produce a processed result, are collectively referred to as the result’s “pedigree”. Data and the modules themselves are all available for download and can be run manually as desktop modules (independent of the Engine).

1.3 Guiding Principles

For shot data analysis in NIF, developers should adhere to the following guiding principles that are listed below and were established in our initial requirements (see [1]).

- Analysis functions will be published and “transparent” using known calibration measurements; i.e., everyone will know what steps were performed to arrive at the results, and they have access to those results.
- The data will be tracked and controlled so that the analysis pedigree is known and the visualization team can display the results. Hand-entered figures or undocumented calculations will be strongly discouraged. The data will be stored in CMS.
- Analysis steps need to be configurable, based on the type of shot and the components of the NIF system that are participating.
- Where possible, uncertainty (error bounds) and data quality metrics will be measured/calculated, published and archived with the data.
- Analysis results will be made available in a time period so they can be applied to the next decisions. The current NIF data analysis requirement is 30 minutes for “quick looks” of shot data.
- Comparisons to off-line data or simulations will be supported. These modeled or simulated data will be made available to the analysis software when available, preferably prior to the shot, for comparison purposes; i.e., the expected results (and error bounds) will be compared to measured data as part of the analysis.
- The ability to re-run the analysis of the data, off-line from the shot, using different parameters or analysis methods will be supported. Any data from prior shots will be available for this off-line analysis.
- Likewise, the user will be able to download the analysis algorithms for the purposes of running the analysis software on synthetic or customized data not associated with any shot, such as “what-if” studies and tuning tests.
- Emphasis is on data that are automatically acquired and are required for making shot-related decisions, though other data will be involved as well.

2.0 Setting up an IDL Developer Environment

Application Developers can set up a local IDL development environment as they wish, whether it is on their own machine or a server, using the manuals that come with an IDL license. After installing IDL, go to the NIF Wiki site² <https://nif-wiki.llnl.gov/display/sdi/Downloads> and download the zip file `sda_idl_X.X.X.zip` (where `X.X.X` is the latest version). This file contains IDL utility functions and procedures to handle common tasks (error reporting, message logging, etc.), as well as test functions, a template file (`td_func_template.pro` in Figure 2 and described later), and some common global variable definitions in `td_retcodes.inc` and `td_logdefs.inc`. A test function `td_testenv.pro` can be executed in IDL to demonstrate that the environment is working properly; it generates a log file `td_log_date.txt` in the `TD_LOGFILEPATH` directory. (See Appendix A for more details.) Other useful files in that directory are described in the `README.txt` file, also on the Wiki site.

² Access to the NIF Wiki can be requested at <https://gethelp.llnl.gov> through the Remedy system.

```

#####
;
;           Copyright 2007 University of California
;           Lawrence Livermore National Lab
;           National Ignition Facility
;
; File:      $RCSfile: td_func_template.pro,v $
; Revision Number: $Revision: 1.7 $
; Last Modified: $Date: 2007/09/18 20:54:46 $
;             $Author: glenn21 $
;
; AUTHOR:
;   A. Developer, 5/30/07
;
; PURPOSE:
;   Computes sum and difference of two identically-sized images. This section
;   should contain relevant background information, a description of the routine does,
;   a list of known limitations, and any critical assumptions made about its inputs.
;
; EXAMPLE:
;   a = dist(256)
;   b = shift(a, 128, 128)
;
;   errStatus = td_func_template( $
;     img1 = a, $
;     img2 = b, $
;     fudge_factor = 0.33, $
;     sum_img = a_plus_b, $
;     diff_img = a_minus_b, $
;     quality_flag = goodQual, $
;     errmsg = errmsg )
;
;   if (errStatus ne 0) then $
;     print, errmsg $
;   else tvscl, sum_img
;
; REFERENCES:
;   Some_document.pdf
;
; INPUT PARAMETERS:
;   NAME      TYPE      DESCRIPTION
;   img1      2D array   First image
;   img2      2D array   Second image. Dimensions and type must match img1
;   fudge_factor float   Correction factor.
;   bad_pixels uint      List of bad pixel indices.
;
; OUTPUT PARAMETERS:
;   NAME      TYPE      DESCRIPTION
;   sum_img   2D array   Normalized sum of img1 and img2
;   diff_img  2D array   Difference of img1 and img2
;   quality_flag int     Results quality flag: 1 if good, 0 if bad.
;   errmsg    string     Error description. Empty if errStatus is 0.
;
; RETURNED VALUES:
;   NAME      TYPE      DESCRIPTION
;   errStatus int       0->success, -1->failure
;
; REVISION HISTORY:
;
; NAME      DATE      CHANGE DESCRIPTION
; -----
; aperson33  04/23/07  New file.
; buser7    10/05/07  Added over/underflow error handling.
;
#####

FUNCTION td_func_template, $
  img1 = img1, $
  img2 = img2, $
  fudge_factor = fudge_factor, $
  sum_img = sum_img, $
  diff_img = diff_img, $
  quality_flag=quality_flag, $
  errmsg=errmsg
;
module_name = 'FUNC_TEMPLATE' ; for message logging
@td_retcodes.inc             ; include return code definitions
@td_logdefs.inc              ; include log filter level definitions
errStatus = TD_STATUS_NORMAL ; initialize error status value
quality_flag = TD_QUALITY_BAD ; initialize quality flag
errmsg = ''                  ; initialize returned error message

td_log, '<-- Begin', td_logdefs.INFO, module_name=module_name

```

Used by the CVS Version
Control; DO NOT MODIFY

Header
Comment
Block

Function name &
parameters; Variables
passed as keywords

Store a start message
in the log file

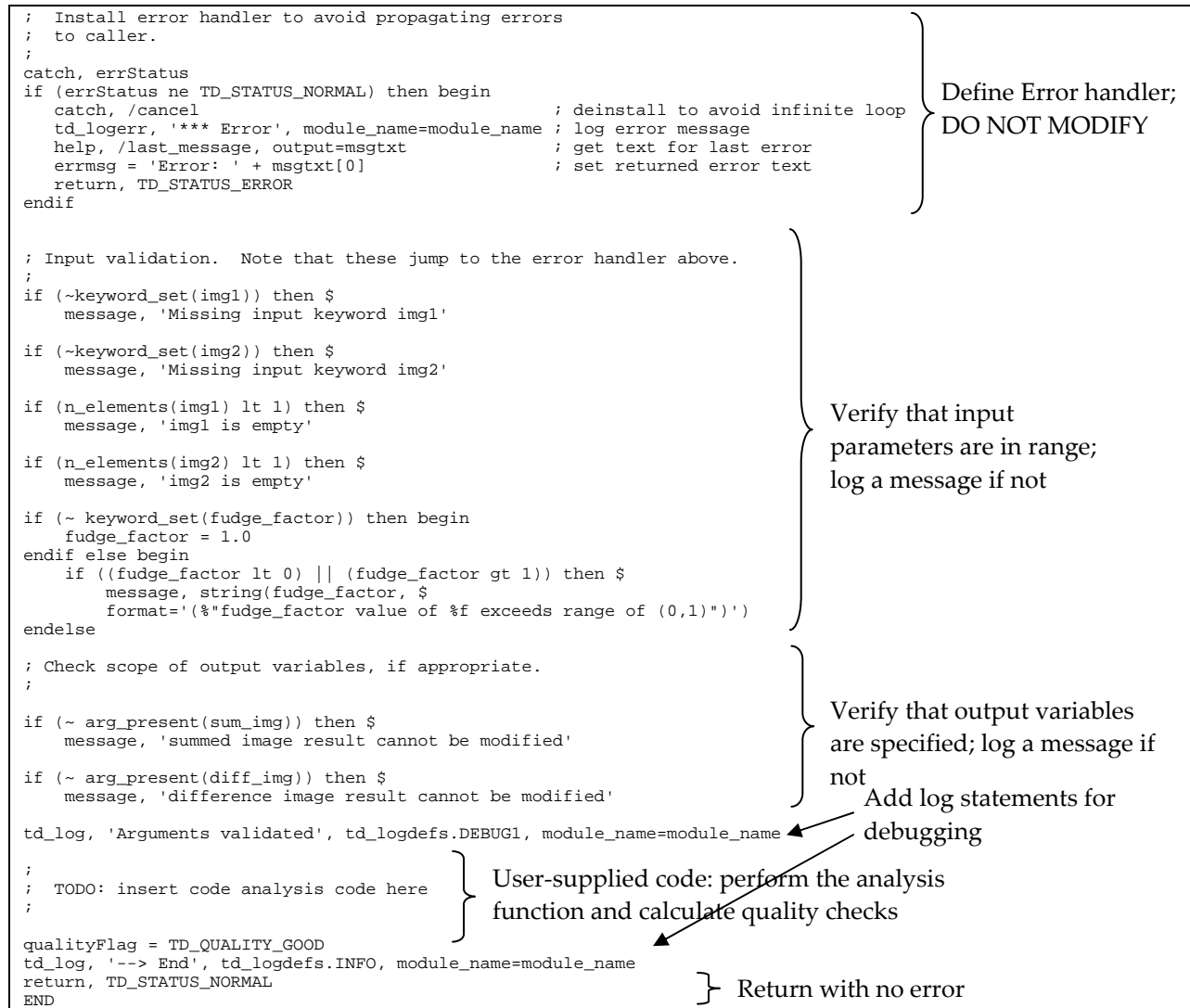


Figure 2. Template IDL Module.

3.0 IDL Analysis Module Guidelines

The Application Developer has many ways to write an analysis module. However, developers should adhere to the following general guidelines.

3.1 Use Functions, not Procedures

The SDA Engine invokes modules as IDL functions, not procedures. The top-level function is expected to return at least one value – an error flag. Internal to the module, procedures may be used. The top-level function file name must match the function name itself. Developers are encouraged to use the symbolic constant `TD_STATUS_NORMAL` (or `TD_STATUS_ERROR`) for return values to indicate success (or failure) as illustrated in the template module.

3.2 Pass Data through the Argument List with Keywords

In order to run automatically under the SDA Engine, the module cannot perform its own interaction with the user or data archive. All data needed (or generated) by the module must be transferred in or out of the top-level function through its argument list and return value; that is, automated modules do not make database calls directly, or use global variables to pass information, and internal variables are not visible to the Engine.

In addition, functions should be called with IDL keyword arguments to avoid errors in the calling sequence. The syntax in IDL argument lists is to enter a keyword name followed by an equal sign ("=") prior to the value to which the keyword should be set. The value can be a constant, an expression, or an IDL named variable. (See the IDL documentation [2] for more details.) By employing keywords in function calls, the argument list can be order independent and the arguments can be easily validated. An example function call using keywords for the fictional function above is given in its header.

```
a = dist(256)
b = shift(a, 128, 128)

errStatus = td_func_template( $
    img1 = a, $
    img2 = b, $
    fudge_factor = 0.33, $
    sum_img = a_plus_b, $
    diff_img = a_minus_b, $
    quality_flag = goodQual, $
    errmsg = errmsg )
```

3.3 Provide Test Code and Data

In the process of developing modules, it is assumed that one or more examples of sample input data will be available for testing the algorithms. The Developer should generate a simple unit-test function that invokes the module with the sample data that produces known results. When development is complete, the sample data and the unit-test function should be made available to the SDA development team along with the module source code. That way, when the module is turned over for configuration management and insertion into the automatic analysis procedures, there are examples that can be used to test their operation.

A unit-testing procedure or function should be named `test_pkg.pro`, where *pkg* is the name of the module package. The unit-test code should invoke the module with the sample data set(s), exercising as much of the module functionality as possible.

3.4 Reduce Duplication

Developers are encouraged to look at the IDL libraries and `td_util` directory for shared functions that can be re-used. Duplication of functionality will lead to maintenance issues, and should therefore be minimized. In the initial version of this directory, there are functions for basic image integrity checking (`td_camera_imagecheck`) and log file utilities (`td_log`).

3.5 Provide Early Error Notification

Modules should verify that required input information is present, and should provide early error notification before lengthy analysis is performed. If input data or calibration values must be within a

known range, for example, the module should check for that range and at least log a message (if not return an error flag and error message).

3.6 Document the Module

Besides the code itself (the .pro file), there are three documents that must be associated with a module: a module requirements document, a module input/output summary spreadsheet, and a module data map.

The module requirements document contains the following information.

- Overview of the module (scope, description, uses)
- Usage – Types of data/instruments for which it is used
- Calibration information needed (if applicable)
- Data flow and a data summary
- Algorithm theory including references, if available.
- Example input data and results, including off-normal data
- Metrics for measuring data quality (that are returned from the module)
- Visualization requirements (high level)
- Testing methods (see Section 4.0 Module Testing)
- Revision and review history

The module input/output summary spreadsheet provides a specific and detailed list of arguments and their properties so that the interface to the module is clearly defined.

The module data map is another spreadsheet that defines, for each *use* of the module, where the input data is to be found and where the output data is to go. For example, an Image Background Correction module may be used for both streak cameras and for static-frame cameras, but their resulting data may be stored in very different places in the database using CMS. The module data map lists the specific data storage locations in the data base for each argument in the input/output summary, and for each use of the module.

All three of the above documents are under revision control for those modules maintained by the SDA development team. Examples of all of these documents are available on the NIF wiki site, or by request.

3.7 Return Metrics

SDA Modules should return metrics about the results—for example, values that indicate how well the algorithm performed (a “quality flag”) and the statistical error bounds (one-sigma) for the results when possible. The file `td_retcodes.inc` in the `td_util` package contains the following definitions:

- `td_quality=1` Good quality output data
- `td_quality=0` Bad quality; the output data may have a problem

The other metrics and return parameters should provide more information about the severity of the problem and the performance of the algorithm. For multi-step modules, separate quality flags may be needed for each step, with the final quality flag being the product of each one.

3.8 Follow Naming Conventions

A naming convention has been established to help users describe, organize and find modules. Since IDL has one global namespace, these conventions will also help reduce naming conflicts.

- Lower-case letters, numbers and underscores – All functions should start with a letter and contain only letters, numbers and the underscore character (as a separator). No other special characters should be used.
- Lower case – All functions and file names should only use lower-case letters (no upper or mixed cases) to facilitate operation on diverse platforms.
- Routines for general shot data domains should have the area tag at the beginning of the name as shown:

lp	laser performance
oi	optics inspection
td	target diagnostics
- Specific routines have both the domain tag and the sub-domain tag in the name, followed by a descriptive phrase of the routine. For example, sample routine names for target diagnostics are:

td_sxi_pattern_find
td_img_flatfield
td_img_deadpixfix

3.9 Use the Standard Layout

Each module should contain the programming elements and comment blocks shown in the template (Figure 2) and listed below:

- Header comment block – Includes copyright notice, version control information, author, purpose of the code, references, input and output parameters (with data types and description), returned values, assumptions, library dependencies, example usage and revision history. The version-control block should be left unchanged as the revision management system (CVS) uses those fields to track changes (see Section 5.0 Revision Control).
- Function (or procedure) name and parameters – Include the returned error status and error message.
- Logging definitions – The command “@td_logdefs.inc” defines the message logging flags to be used.
- Return code definitions – The command “@td_retcodes.inc” defines the symbolic constants for returned error status and results quality flags.
- Error handler – Establishes a common error handler that forces a message and graceful exit, while avoiding infinite loops.
- Parameter verification – A block that checks the validity of parameters, providing clear error messages in case of problems, is good programming practice.
- User-supplied algorithm – Performs the operations stated in the purpose. Liberal comments are always recommended.
- Return the error flag in all cases with the following values (from td_retcodes.inc)

○ TD_STATUS_NORMAL	No error
○ TD_STATUS_ERROR	Error

A module file can also contain procedures or functions that are only used by the main function. Comments should be employed throughout the file to document the module.

Messages of various types (errors, warnings, information, debug) should also be used to aid development and maintenance (see Section 3.12 Log Messages).

3.10 Use IDL Structures as Needed

IDL data structures or classes can be used in SDA modules, and their use is encouraged. Structures help organize the passing of values from one function to another, simplify input parameter checking, accommodate future modification request, and increase maintainability. General and detailed instrument structure elements and types will be determined at a later date.

3.11 Avoid Global Variables or Common Blocks

Avoid the use of global variables or common blocks that pass data implicitly. Otherwise, behavior of a function may not be apparent from its interface. (The message logging variables in Section 5.13 are a special case since a single point of global access is required.) Also, if there are dependencies on foreign procedures or libraries, they should be noted in the header and care should be taken to avoid duplication of names across all procedures in use.

3.12 Do Not Use Display Commands

The Developer is discouraged from using display output commands such as `tv`, `plot`, `device`, or graphical widgets such as `iimage` or `itool` within the normal path of execution. This is because the Analysis Engine framework does not support graphical screen output. Instead, the calling routine, or a separate visualization tool, should be employed to display data results from the module.

3.13 Log Messages

To aid debugging, a message-logging package has been implemented so that text output from modules can be systematically stored in a central location. The message logging mechanism is not intended for analysis results storage.

Messages are sent to a common log file established by the SDA Engine (or calling procedure). The user can set a “messaging level” that establishes at run-time whether messages are written or not using the `td_set_log_filter` routine before running an analysis module. We define the following message levels:

- Error – Causes the module to return immediately with error flag set
- Warning – Serious problem that does not cause a halt (or error)
- Info – No problems; report that a line in the code is reached at a certain time
- Debug – Provide additional information for debugging purposes. We define three levels of debug logging (DEBUG1, DEBUG2, DEBUG3) that output successively more information. The level DEBUG_ALL turns on all the above messaging.

It is up to the developer to decide where and when messages should be sent to the log file and at what level. Problems are easier to diagnose if frequent messages are logged that contain incremental results or progress indicators. On the other hand, it is easy to over-comment; e.g., messages reported in a tight loop can flood the log, causing confusion and, in extreme cases, adversely affecting throughput and disk space.

The “`td_log`” procedure is the function for writing all messages using the following construct (also demonstrated in Figure 2):

```
td_log 'msg', msg_level, [module_name='routine_idntifer']
```


where `'msg'` is a string that will be written to the log file, and `msg_level` is one of the following global values corresponding to the above messaging levels:

- `td_logdefs.ERROR`
- `td_logdefs.WARNING`
- `td_logdefs.INFO`
- `td_logdefs.DEBUG1`
- `td_logdefs.DEBUG2`
- `td_logdefs.DEBUG3`

The `'msg'` string should be descriptive enough to provide the reader with real information about the process. See examples of message logging in the provided code template.

Writing to “standard output” can be done also. During automatic operation, the Analysis Engine captures anything written to standard output and saves it in a log file along with the other log messages. To maintain consistency of messages, using the `td_log` mechanism is preferred.

3.14 Use Supported Procedures

As with all commercial and open source software packages, new releases of, say, IDL will require review and testing of existing code—and changes are likely to be needed to maintain compatibility. To reduce problems of obsolete functions and other effects, Developers should use clearly-supported functions and procedures that are not expected to change in the future. Backward-compatibility tests will be performed on all SDA software before a new version of commercial software is employed.

4.0 Module Testing

Before a module is accepted for use, it must go through three types of testing – unit test (stand-alone), integration test (in the SDA Engine), and performance verification. The testing and documentation requirements, described in this section, are the same whether developed by RS or the SDA team.

4.1 Unit Test

As described in Section 3.3 Provide Test Code and Data, the Developer will supply a unit test function (`test_pkg.pro`), with supporting data, which can be performed in a manual or stand-alone mode. The input data may originate from anywhere—previous experiments or simulated data from the RS or simply made up—as long as it exercises the module functions, including common error modes. This type of testing can be simply a function to check input/output data types and graceful error handling. It only verifies that the resultant data is reasonable, not the accuracy or precision of the results.

4.2 Integration Test (Verification)

This type of testing verifies the resulting data of an individual module or a group of modules that are invoked by the SDA Engine. It requires that the test data be located in the central data archive where the SDA Engine can retrieve the data, provide data to the module and store resulting data to the archive.

The integration tests only validate the module operation meets requirements given the (limited) set of test data.

Prior to code release, a control script will systematically run each test data set in the central archive through the module and compare the results to stored values. Both normal and off-normal data should be included in the integration tests. The archive of test data will increase in size over time as new test cases are discovered.

4.3 Performance Validation

In conjunction with the scientific expert (the RS), the module developer will perform a series of tests, with many types of inputs and parameters, to validate the scientific merit of the output results. Each new set of input data, with its own uncertainties and noise sources, can cause unexpected numerical results instabilities in the module algorithms. Performance validation is a process that continues throughout the useful life of the code.

5.0 Revision Control

For Analysis Modules that are run automatically, the SDA development team is managing the software configuration through a standard version control system called CVS. A separate document [5] describes the CVS structure and methodology for the SDA version control. The main goals are to

1. Independently manage and release groups of source files in Analysis Packages – Example packages are common utilities, camera corrections, SXI analysis, Dante analysis, etc.
2. Support the establishment of analysis pedigree – Keep track of the code version that generates a particular analysis result, in addition to the raw data, calibration and other parameters that affect the data.
3. Support dependency analysis – Be able to determine which results are potentially affected when a module group is revised, and initiate analysis re-runs if needed. This feature may also be used to guide testing.
4. Maintain transparency – Allow all who are interested in NIF results be able to see how those results were achieved at every step.

The revision control procedure described in [5] has the following general outline that will be refined over time. It applies mainly to code that is managed by the SDA development team.

- Change requests – Requests to make changes can be e-mailed to any member of the SDA development team, where they are assigned and given a priority.
- Design the repair – The SDA Developer will design the modifications needed and review them with appropriate personnel (RS and/or SDA team).
- Implementation – The changes are made and tested on a unit basis in the SDA development environment. The results are reviewed by the SDA team.
- Documentation changes – Both the module requirements document and the internal code documentation must be updated.
- Regression test – Deliver data and test procedures for automatic testing.
- Release to Production.

6.0 Summary

This document presents the Shot Data Analysis module development guidelines for IDL codes. For modules that are to be used in automatic analysis following a NIF shot, adherence to these guidelines will be required. Templates and common procedures are available in the NIF Wiki (<https://nif-wiki.llnl.gov/display/sdi/Downloads>) to aid the developer.

References

1. "Shot Data Analysis Conceptual Design Review", NIF Internal Document, <https://nif-wiki.llnl.gov/display/sdi/CDR>, Dec 2006.
2. Gumley, L. E., *Practical IDL Programming*, Morgan Kaufmann, 2001.
3. Bettenhausen, R. C., "Shot Analysis System Conceptual Design", v1.1, [WBS5 Analysis FW Concept Sys Des v1.1.doc](#), Dec 2006.
4. Bettenhausen, R. C. and Mak, R., "Shot Analysis System Driver Design", v1.0.4, in progress, June 2007.
5. Glenn, S., "Shot Analysis Module (IDL) Configuration Management", in progress, 2007.

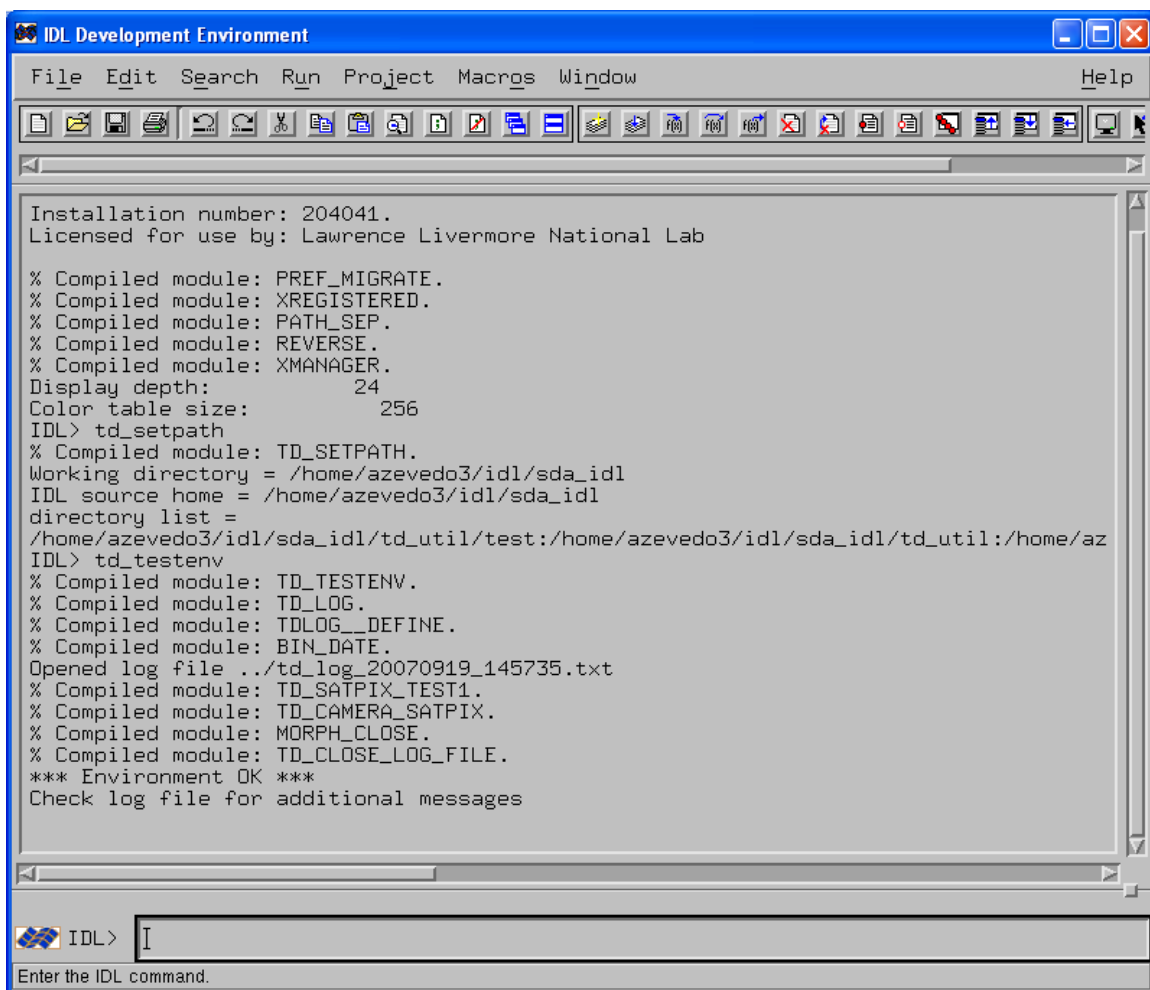
Revision Log:

Rev.	Date	Pages	Brief Description of Revision
1.0	11/22/2006	All	Initial Version (A. Lopez)
1.1	11/30/2006	All	Format Change (A. Warrick)
	11/30/2006	1	Introduction (R. Beeler)
	11/30/2006	1	Introduction, module/procedure (A. Lopez, R. Beeler, SDA Team)
	12/01/2006	2	Add tag to section 2 (A. Lopez)
1.2	12/07/2006	4-5	Module acceptance, backward compatibility (A. Lopez)
1.3	12/14/2006	4-5	Backward compatibility (A. Lopez)
2.0	3/21/2007	1-3	Expand on module explanation, add module and procedure section (A. Warrick)
3.0	7/10/2007	All	Expand in all areas for general use (S. Azevedo)
3.1	7/16/2007	All	Include templates (S. Azevedo)
3.2	8/27/2007	All	Major changes (S. Azevedo)
3.3	9/11/2007	All	Refinements (S. Azevedo)
3.4	9/28/2007	All	Final Templates (S. Azevedo)
3.5	10/1/2007	9,13-14	Log info; Added Appendix A (S. Azevedo, S. Glenn)

Appendix A

Example module: `td_testenv.pro`

In Section 2.0 Setting up an IDL Developer Environment, we introduced a sample IDL file that can be used to test the user's environment. Once the environment is established, load the `td_testenv.pro` file into IDL and type the two commands highlighted in the figure below – `td_setpath` to set the directory paths and `td_testenv` to perform a simple saturated-pixel correction and write information to the log file. The screen output should appear similar to Figure A-1.



```
IDL Development Environment
File Edit Search Run Project Macros Window Help

Installation number: 204041.
Licensed for use by: Lawrence Livermore National Lab

% Compiled module: PREF_MIGRATE.
% Compiled module: XREGISTERED.
% Compiled module: PATH_SEP.
% Compiled module: REVERSE.
% Compiled module: XMANAGER.
Display depth:      24
Color table size:   256
IDL> td_setpath
% Compiled module: TD_SETPATH.
Working directory = /home/azevedo3/idl/sda_idl
IDL source home = /home/azevedo3/idl/sda_idl
directory list =
/home/azevedo3/idl/sda_idl/td_util/test:/home/azevedo3/idl/sda_idl/td_util:/home/az
IDL> td_testenv
% Compiled module: TD_TESTENV.
% Compiled module: TD_LOG.
% Compiled module: TDLOG__DEFINE.
% Compiled module: BIN_DATE.
Opened log file ../td_log_20070919_145735.txt
% Compiled module: TD_SATPIX_TEST1.
% Compiled module: TD_CAMERA_SATPIX.
% Compiled module: MORPH_CLOSE.
% Compiled module: TD_CLOSE_LOG_FILE.
*** Environment OK ***
Check log file for additional messages

IDL> |
Enter the IDL command.
```

Figure A-1. IDL screen output for `td_testenv`.

The module will generate a log file that can be found in the TD_LOGFILEPATH directory, where TD_LOGFILEPATH is an environment variable that is set in IDL (default is the current working directory). Contents of the log file should resemble the output below.

```

2007-09-28 03:01:50 10 TESTENV          <-- Begin
2007-09-28 03:01:50 10 SATPIX_TEST1     **** Begin Saturated Pixel Test 1
2007-09-28 03:01:50 10 SATPIX          <-- Begin
2007-09-28 03:01:50 03 SATPIX          Arguments validated
2007-09-28 03:01:50 00 SATPIX          image min, max, median intensity = 0.000000,
255.000000, 67.000000
2007-09-28 03:01:50 00 SATPIX          bright thresh1, thresh2, n1, n2 = 251, 246, 4207, 4207
2007-09-28 03:01:50 00 SATPIX          bright thresh1, thresh2, n1, n2 = 255, 250, 3976, 4207
2007-09-28 03:01:50 03 SATPIX          Found 1 bright-saturated lines with minimum length 77
2007-09-28 03:01:50 03 SATPIX          Finished bright saturation analysis
2007-09-28 03:01:50 00 SATPIX          dark thresh1, thresh2, n1, n2 = 1, 6, 140, 137
2007-09-28 03:01:50 00 SATPIX          dark thresh1, thresh2, n1, n2 = 0, 5, 115, 137
2007-09-28 03:01:50 03 SATPIX          Found 0 dark-saturated lines with minimum length 77
2007-09-28 03:01:50 10 TD_CAMERA_SATPIX --> End
2007-09-28 03:01:50 10 SATPIX_TEST1     quality = 1
2007-09-28 03:01:50 10 SATPIX_TEST1     bright_frac= 0.064194, 0.060669
2007-09-28 03:01:50 10 SATPIX_TEST1     dark_frac 0, 0.001755,
2007-09-28 03:01:50 10 SATPIX_TEST1     bright_sens= 0.000000, -0.058099
2007-09-28 03:01:50 10 SATPIX_TEST1     dark_sens 0, 0.191304,
2007-09-28 03:01:50 10 SATPIX_TEST1     **** Completed Saturated Pixel Test 1
2007-09-28 03:01:50 10 TESTENV          --> End

```

Figure A-2. Output log for td_testenv.